XDP closer integration with network stack

Jesper Dangaard Brouer
Kernel Developer
Red Hat

Kernel Recipes Conf
Paris, Sep 2019

Overview: What will you learn?

Audience at Kernel Recipes: Other kernel developers

- But for different kernel sub-systems
- Learn about: kernel network data structures

Hopefully you already heard about XDP (eXpress Data Path) ? ? ?

- I'll explain why network sub-system created this...

Future crazy ideas to extend XDP

- (hopefully) in a way that cooperate more with kernel netstack

Give you an easy way to try out XDP and BPF on your laptop

Why was XDP needed?

This was about the kernel networking stack staying relevant

- For emerging use-cases and areas

Linux networking stack assumes layer L4-L7 delivery

- Obviously slow when compared to L2-L3 kernel-bypass solutions

XDP operate at layers L2-L3

- Shows same performance as these L2-L3 kernel-bypass solutions

The networking OSI layer model:

- L2=Ethernet
- L3=IPv4/IPv6
- L4=TCP/UDP
- L7=Applications

What is XDP?

What kind of monster did we create with XDP?!?

XDP (eXpress Data Path) is a Linux in-kernel fast-path

- New programmable layer in-front of traditional network stack
  - Read, modify, drop, redirect or pass
- For L2-L3 use-cases: seeing x10 performance improvements!
- Can accelerate in-kernel L2-L3 use-cases (e.g. forwarding)

What is AF_XDP? (the Address Family XDP socket)

- Hybrid kernel-bypass facility
- Delivers raw L2 frames into userspace (in SPSC queue)

AF_XDP: Used for kernel-bypass?!?

Did you really say, this can be used for bypassing the Linux kernel netstack ?

Sure, build in freedom for kernel-bypass via AF_XDP

- DPDK already have a Poll-Mode driver for AF_XDP

Why is this better, than other (bypass) solutions?

- Flexible sharing of NIC resources, NIC still avail to netstack
- XDP/eBPF prog filters packets using XDP_REDIRECT into AF_XDP socket
  - Move selective frames out of kernel, no need to reinject
- Leverages existing kernel infrastructure, eco-system and market position

How can XDP be (ab)used?

Used or abused?

- Freedom re-implement everything (when bypassing the kernel)
- Or freedom to shoot yourself in the foot?

Simple view on how XDP gains speed

XDP speed gains comes from

- Avoiding memory allocations
  - no SKB allocations and no-init (memset zero 4 cache-lines)
- Bulk processing of frames
- Very early access to frame (in driver code after DMA sync)
- Ability to skip (large parts) of kernel code

Skipping code: Efficient optimization

Skipping code: Imply skipping features provided by network stack

- Gave users freedom to e.g. skip netfilter or route-lookup
- But users have to re-implement features they actually needed
  - Sometimes cumbersome via BPF-maps

Avoid re-implement features:

- Evolve XDP via BPF-helpers

Evolving XDP via BPF-helpers

We should encourage adding helpers instead of duplicating data in BPF maps

Think of XDP as a software offload layer for the kernel netstack

- Simply setup and use the Linux netstack, but accelerate parts of it with XDP

IP routing good example: Access routing table from XDP via BPF helpers (v4.18)

- Let Linux handle routing (daemons) and neighbour/ARP lookups
- Talk at LPC-2018 (David Ahern): Leveraging Kernel Tables with XDP

Obvious next target: Bridge lookup helper

- Like IP routing: transparent XDP acceleration of bridge forwarding
  - Fallback for ARP lookups, flooding etc.
- Huge potential performance boost for Linux bridge use cases!

Understand networking packet data structures

To understand next slides and (XDP) kernel networking

- Need to know difference between some struct's
- Used for describing and pointing to actual packet data

Fundamental struct’s

The struct’s describing data-frame at different levels

- `sk_buff` : Good old SKB, allocated from SLAB/kmem_cache (4 cachelines)
- `xdp_buff` : Used by BPF XDP-prog, allocated on call stack
- `xdp_frame`: xdp_buff info state compressed, used by XDP-redirect
  - No allocation, placed in top of data-frame (currently 32 bytes)
- HW specific “descriptor” with info and pointer to (DMA) data buffer
  - contains HW-offloads (see later) that driver transfers to SKB

Exotic details

- `skb_shared_info` : placed inside data-frame (at end), e.g. GRO multi-frame
- `xdp_rxq_info` : Static per RX queue info

Evolving XDP: Future ideas

Warning: Next slides about crazy future ideas

- This stuff might never get implemented!

Move SKB allocations out of NIC drivers

Goal: Simplify driver, via creating SKB inside network-core code

- Happens today via `xdp_frame` in both veth and cpumap
- (Slight hickup: Max frame size unknown, thus lie about skb->truesize)

Issue: SKB's created this way are lacking HW-offloads like:

- HW checksum info (for `skb->ip_summed` + `skb->csum`)
- HW RX hash (`skb_set_hash(hash, type)`)
- (these are almost always needed... tempted to extend `xdp_frame`)

Other HW-offloads

Other existing offloads, used by SKBs, but not always enabled

- VLAN (`__vlan_hwaccel_put_tag()`)
- RX timestamp
  - HW `skb_hwtstamps()` (stored in skb_shared_info)
  - Earlier XDP software timestamp (for `skb->tstamp`)
- RX mark (`skb->mark` supported by mlx5)

Other potential offloads, which hardware can do (but not used by SKB):

- Unique u64 flow identifier key (mlx5 HW)
- Higher-level protocol header offsets
  - RSS-hash can deduce e.g. IPv4/TCP (as frag not marked as TCP)
  - But NIC HW have full parse info avail

Blocked by missing HW-offloads

SKB alloc outside NIC driver, blocked by missing HW-offloads.
The GOAL is to come-up with a Generic Offload Abstraction Layer...

Generic and dynamic way to transfer HW-offload info

- Only enable info when needed
- Both made available for SKB creation and XDP programs

The big questions are:

- Where to store this information?
- How to make it dynamic?
- What else are we missing?

Storing generic offload-info

Where to store generic offload-info?

- To avoid allocation use packet/frame data area
  - (1) Extend xdp_frame: imply top of frame head-room
  - (2) Use XDP meta-data area: located in-front of payload start
  - (3) Use tail-room (frame-end): Already used for skb_shared_info GRO

No choice done yet...

Dynamic generic offload-info

Next challenge: How to make this dynamic?

- Each driver have own format for HW descriptor
- Hopefully BTF can help here?

Drivers could export BTF description of offload-info area

- BPF prog wanting to use area, must have matching BTF
- But how can kernel-code use BTF desc and transfer to SKB fields?

Dependency: Handling multi-frame packets

SKB alloc outside NIC driver, ALSO need XDP multi-frame handling

Multi-frame packets have several use-cases

- Jumbo-frames
- TSO (TCP Segmentation Offload)
- Header split, (L4) headers in first segment, (L7) payload in next

XDP need answer/solution for multi-frame packets

- To fully move SKB alloc+setup out of NIC drivers
  - (SKB use skb_shared_info area to store info on multi-frames)
- Design idea/proposal in XDP-project: xdp-multi-buffer01-design.org

Fun with xdp_frame before SKB alloc

After SKB alloc gets moved out of drivers

- What can we now create of crazy stuff?!?

General idea for xdp_frame handling

Idea: Use xdp_frame for some fast-paths

- E.g. forwarding could be accelerated (non localhost deliver)
- Fall-back: Create SKB for slow(er) path
- Lots of work: adjust functions to work without SKBs

New (L2) layer with xdp_frame?

Could update netstack (L2) RX-handler to handle xdp_frame packets?

- Bridging
- Macvlan, ipvlan, macvtap
- Bond + Team
- OpenVSwitch (OVS)

Likely: Need new L2 RX-handler layer (?)

- To support kernels evolutionary development model
  - (cannot update every user at once, plus out-of-tree users)

Transport layer XDP

XDP operate at L2 (Eth) or L3 (IP)

- Tom Herbert (coined XDP) proposed Transport-layer XDP = L4 (TCP/UDP)
  - To gain performance it need to operate on xdp_frame's
  - For many fast-path TCP/UDP use-cases, SKB is pure overhead
    - Much simpler xdp_frame will be sufficient
    - Let special cases fall-through, alloc+init full SKB

More practical

People complain XDP and eBPF is hard to use!?

XDP-tutorial

Ready to use: XDP Hands-On Tutorial

- Basically a full build and testlab environment

Simply git clone and run make:

- https://github.com/xdp-project/xdp-tutorial

Testlab works on your (Linux) laptop

- via creating `veth` device and network namespace

The missing transmit side

XDP is currently only an RX-hook

- We want to change that! - but it’s (also) a big task

Tell people to use TC-BPF egress hook, we can do better...

The missing XDP transmit hook

A real XDP TX hook is needed, for several reasons

- Creates symmetry (RX and TX hooks)
- XDP-redirect need push-back/flow-control mechanism
  - Too easily overflow (HW) TX-queues (as qdisc-layer bypassed)
  - Be careful: don't re-implement full qdisc layer

Should also run after normal qdisc layer TX-queues

- Reason: Like BQL, can choose when qdisc is activated
  - Allows to implement BQL (Byte-Queue-Limit) as eBPF
- Driver TX-q based on xdp_frame, allow SKB to be released earlier

End: Summary

Kernel now have eBPF programmable network fast-path

- that can now compete with kernel-bypass speeds

Not finished: Still lots of development work ahead

- Need to cooperate more with kernel netstack
  - Create BPF-helpers to access kernel tables
  - How far can we take this: SKBs outside drivers? realistic goal?

XDP-project coordination:

- https://github.com/xdp-project/xdp-project